ANALYZING SOFTWARE BEHAVIOR

Background

This application relates generally to analyzing software and more particularly to analyzing software behavior.

Computer systems generally include a central processing unit, a memory system, and a data storage system. An enterprise data storage system (EDSS), such as the Symmetrix Enterprise Storage Platform (ESP) by EMC Corp., is a versatile data storage system having the connectivity and functionality to simultaneously provide storage services to different types of host computers (e.g., mainframes and open system hosts). A large number of main, physical storage devices (e.g., an array of disk devices) may be used by an EDSS to provide data storage for several hosts. The EDSS storage system is typically connected to the associated host computers via dedicated cabling or a network. Such a model allows for the sharing of centralized data among many users and also allows a single point of maintenance for the storage functions associated with the many computer systems.

Disk drive systems continue to grow in size and sophistication. These systems can typically include many large disk drive units controlled by a complex, multi-tasking, disk drive controller such as the EMC Symmetrix disk drive controller by EMC Corp. A large scale disk drive system can typically receive commands from a number of host computers and can control a number of disk drive mass storage devices.

As these systems increase in complexity, so does the user's reliance upon the systems, for fast and reliable access, recovery, and storage of data. Accordingly, the user typically uses data throughput and speed of response as primary criteria for evaluating

performance of the disk drive systems. As a result, mass storage devices and the controllers that drive them have become quite sophisticated in efforts to improve command response time. Systems such as the EMC Symmetrix disk drive controller system thus incorporate a large cache memory, and other techniques to improve the system throughput.

As the systems grow in complexity, interrupting failures at either the disk drive or at the controller level become increasingly undesirable. As a result, systems have become more reliable and the mean time between failures continues to increase. Nevertheless, it is more than an inconvenience to the user if the disk drive system goes "down" or off-line; even if the problem is corrected relatively quickly, i.e., within hours. The resulting lost time adversely affects not only system throughput performance, but user application performance. Further, the user is typically not concerned whether it is a physical disk drive, or its controller that fails; it is the inconvenience and failure of the system as a whole that causes user difficulties.

Many disk drive systems, such as the EMC Symmetrix disk drive system, rely upon standardized buses to connect the host computer to the controller, and to connect the controller and the disk drive elements. Thus, should the disk drive controller connected to the bus fail, the entire system, as seen by the host computer, may fail and the result is, as noted above, unacceptable to the user.

As computer systems become more complex, and as businesses rely more upon their computer systems, any performance problem is troublesome, and a performance problem that requires the system to shut down becomes a major and potentially disastrous event.

A failure of or decrease in input/output behavior in, for example, a memory system, could become a bottleneck to efficiency and throughput in the overall operation of the system. Thus, much effort and customer engineering has been directed to being able to resolve problems that occur in the input/output process. A problem may be addressed by taking over the system having the performance problem, determining precisely where and what the problem is by recreating the problem at the customer site, for example by running the applications and data that led to the problem, and then resolving the problem.

Such methods of problem isolation and correction accordingly require the customer's system to be off-line for a period of time, and further can require intensive customer engineer time at the customer site. Aside from being relatively costly in customer time, this method of solving a performance issue can further adversely affect the customer's operations.

Summary

Software behavior, such as software performance, software calling sequence, or software memory utilization, is analyzed in a disk drive controller that has a processor executing computer software stored in a memory communicating with the processor via a local bus. In at least one aspect of the invention, in the controller, computer executable program code is identified that includes a set of computer executable program instructions for recording analytical data for at least a subset of the computer executable program code. The set of computer executable program instructions is disabled from executing. Without halting execution of the computer executable program code, execution of the set of computer executable program instructions is enabled.

In at least another aspect of the invention, in the controller, analytical software is identified for recording analytical data of software execution, and software source code is identified that does not include the analytical software. Computer executable program code is derived from the software source code and the analytical software. The computer executable program code includes computer executable instructions for recording analytical data for at least a subset of the computer executable program code. The computer executable instructions are inactive and are capable of being activated during execution of the computer executable program code.

In at least another aspect of the invention, computer executable program code is identified that includes at least one computer executable program instruction causing execution of analytical program instructions to be avoided. The analytical program instructions cause recording of analytical data for at least a subset of the computer executable program code. Without halting execution of the computer executable program code, a change is performed directed to the computer executable program instruction to allow execution of the analytical program instructions.

Implementations of the invention may provide one or more of the following advantages. A profiling tool can be provided that operates with little or no overhead and with little or no performance penalty relative to ordinary operation of the profiled program code. Profiling of all or part of a set of program code can be enabled or disabled while the program code is in use. Highly effective profiling can be accomplished in a barebones profiling environment. Profiling analysis can be performed with little or no changes to the profiled program code. Multiple different profiling sessions may be executed rapidly by exercising a user interface to designate portions of program code for

profiling. A profiling compatible version of program code can be provided that also serves as a production version of the program code, which can facilitate debugging efforts, particularly where reproducing an identified problem is important or necessary to finding a solution. The program code can be arranged so that only a small layer of profiling program code is added for each function that is rendered accessible to profiling, which can facilitate the efficient use of program storage resources. Statistical information resulting from profiling can be retrieved and displayed at various times, including while the profiled program code is executing. Data resulting from profiling can be retrieved quickly without delays that are typically associated with disk access times.

Other advantages and features will become apparent from the following description, including the drawings, and from the claims.

Brief Description of the Drawings

Figs. 1-2 are block diagrams of computer systems having software behavior analysis.

Figs. 3-5 are illustrations of data structures used in the systems of Figs. 1-2.

Figs. 6, 10-15 are flow diagrams of procedures used in the systems of Figs. 1-2.

Figs. 7-9 are illustrations of program code used in the systems of Figs. 1-2.

Figs. 16A-20 are illustrations of user interface presentations used with the systems of Figs. 1-2.

Figs. 21-28 are illustrations of test data of the systems of Figs. 1-2.

Detailed Description

A "Profiler" system described below provides highly effective software behavior analysis capabilities, particularly in a fault-tolerant, shared bus multi-processor, intelligent mass storage environment that has restrictive timing constraints and that relies on efficient use of computing and data storage resources. In a specific implementation, a small amount of Profiler program code is added to computer software at compile time, which Profiler program code may be modified before or during execution of the computer software to cause execution behavior data (e.g., timing data) to be gathered and recorded for all or a portion of the computer software. As described below, individual functions of the computer software are provided with respective program code headers to allow one or more of the individual functions to be selected or de-selected for behavior analysis ("profiling"), even where a profiled function has a parent function (the function that called the profiled function) or a child function (a function called by the profiled function) or both parent and child functions.

The system can be used for collecting behavior data, program flow information (the sequence of function calls executed by a program), and program code coverage information (log identifying executed functions and unexecuted functions). Such use is helpful during software testing and can be made with minimal disruption to production code. For example, in code coverage mode, profiling is disabled for a particular routine once the routine has been executed, such that the overhead of having profiling enabled is minimal and diminishes with time. Thus, in code coverage mode, each routine is counted once at most, so that, for each routine, the result indicates only whether or not the routine was executed at all.

The gathered execution behavior data is stored in one or more table data structures in which an entry is provided for each profiled function, as described below. The table data structures reside in local semiconductor memory that is accessible at high speed by the local processor, e.g., via a data bus that is used primarily or exclusively for data transfers between the local semiconductor memory and the local processor. Profiling results are derived from the table data structures and are presented to the user, e.g., on a computer screen. Profiling statistics are updated in real time as the profiled functions are executed.

Since individual functions can be selected or de-selected for profiling, the Profiler system can be used in a narrowing, iterative search to help pinpoint the function that is the source of an identified program execution problem, e.g., a program bug causing an excessive delay or improper behavior. For example, in an initial round in the iteration, a broad range of functions can be selected for profiling and analysis, and only increasingly likely candidates are selected in subsequent rounds for profiling and analysis. In particular, when a function having many child functions is identified as a likely candidate source of a problem, the function's child functions may be selected or deselected one by one until the location of the problem is identified more specifically.

Referring first to Fig. 1, a storage system 14 is shown to include a plurality of host controllers 21a-21n which are, according to a specific implementation, coupled alternately to buses 22 and 23. Each host controller 21a-21n is responsible for managing the communication between its associated attached host computers and storage system 14. The host controllers of the preferred embodiment may include one or more central processing units ("CPUs" or "processors"). The controller CPUs operate under program

control to manage the flow of data between an associated host computer and the storage system 14.

Also coupled alternately to buses 22 and 23 are a plurality of disk controllers 25a-25n. Controllers 25a-25n are here similar in construction to controllers 21a-21n. That is, each includes at least one CPU configured to operate under control of a program loaded into an associate CPU program memory. Coupled to each disk controller is an array of storage devices which as shown here may be magnetic disk devices. Like the host controllers described above, each disk controller is responsible for managing the communications between its associated array of disk drives and the host controllers or global memory 24 of storage system 14.

As illustrated in Fig. 2, the storage system 14 may include a plurality of logical volumes 716a, 716b, . . . , 716n. Each logical volume may be representative of one or more physical disk drive elements (e.g., magnetic disk devices). Interconnecting the host computers and the logical volumes is the disk drive controller 25. Disk drive controller 25 receives memory commands, e.g., read and write commands, from host computers 712 over a bus 719, which may operate, for example, in accordance with a Small Computer System Interface (SCSI) protocol or a Fibre Channel (FC) protocol. Disk drive controller 25 includes a processor 721 serving as the CPU for executing computer instructions of program code. The computer instructions are stored in a memory 723 (e.g., EEPROM) connected to processor 721 via an internal bus 725. Processor 721 may also interact with another local memory 777 that stores data structures as described below. Disk drive controller 25 delivers data associated with memory commands to or from an appropriate

one of logical volumes 716a, 716b, . . . 716n over a second bus 720 which, for example, operates in accordance with a SCSI or FC protocol.

In a typical configuration, controller 25 is also connected to a console computer 722 through a connecting bus 724. Console PC 722 is used for maintenance and access to the controller and can be employed to set parameters of and/or to initiate maintenance commands to the controller, under user control, as described below.

Each host controller 21 (Fig. 1) may have an internal computing architecture similar to that of disk controller 25.

In general, system 14 may have some or all of the characteristics of one or more storage systems described in U.S. Patent No. 6,161,216, which is incorporated by reference.

With reference to Fig. 2, each controller operates under program control in accordance with software 110 (which may be called "microcode", and which may be or include firmware) stored in a memory such as memory 723. It is desirable to analyze the performance of software 110 during execution. One analysis technique ("Event Trace") provides for tracing a particular event during execution. It has been recognized that Event Trace slows down the system and cannot provide sufficiently accurate profiling information. In Event Trace, the points in the program code that are traced must be determined when the program code is created. A variation of the Event Trace, named Dynamic Trace, provides for runtime selection of points to be traced, but the number of selectable point is limited (e.g., to 16). Selection of such points requires the programmer to hypothesize about the program execution path. Another analysis technique ("Code Trace") allows for tracing small portions of the code at the instruction level, which can be

excessively intrusive and detrimental to system performance, and cannot be used for monitoring execution behavior.

The Profiler system described below provides a software analysis technique that executes in an efficient manner with little or no change to the overall performance of the system. The profiler can cause profiler logic to be exercised before and after a designated software function is executed, so that analysis information can be recorded pertaining to the execution of the software function.

In a specific embodiment, the Profiler program code resides in the program memory 723 of the controller and is executed on the CPU of the controller. As described below, data structures used by the Profiler include a stack 130 (Fig. 2) named "PROFILER STACK" having frames such as frame "T_SYMPROF_STACK_FRAME" (Fig. 5) and a table 150 (Fig. 2) named "STATS" having entries such as entry "T_SYMPROF_TABLE_ENTRY" (Fig. 4). (Fig. 3 illustrates examples of related global definitions.)

Software logic in the Profiler that interacts with stack 130 and table 150 includes several software routines described below (see Fig. 6): "_mcount", "profiler_entry", "_mcount_exit", and "profiler_exit". Routines "_profiler_entry" and "_profiler_exit" update table 150 statistics for a function. Routine "_mcount" determines whether the profiler is enabled, and is able to modify stack 130 to control the flow of program execution in accordance with the Profiler .

In a specific embodiment, a compiler causes generation of processor executable code ("machine code") in a function header to be executed when the Profiler is disabled. The function header's machine code includes a branch instruction that skips some Profiler

data embedded in the function header. The Profiler is enabled by replacing the branch instruction with a call to "_mcount" as described below. The replacement is performed at the direction of the user communicating with the CPU over a link such as a serial or Ethernet link.

Each software function to be analyzed has an entry in the STATS table 150. In a specific implementation described below, the STATS table includes data for at least several different statistics including total execution time for a parent function including execution of calls made from the parent function to the parent function's child functions, and execution time for the parent function alone ("self-time").

In the specific implementation described below, the PROFILER STACK stack 130 includes information for each profiled function currently executing. The information includes an index into the STATS table 150 for the function, an absolute entry time, an absolute re-entry time (e.g., for when the function is re-entered after execution of a child function is completed), and self-time.

Fig. 6 shows the flow of the execution of routines in a specific embodiment. When a profiled function 210 is to be executed, a header 212 is executed, which causes execution of the "_mcount" routine 214, which causes the "profiler_entry" routine 216 to be executed. Control is returned to the "_mcount" routine, and then the body 218 of the profiled function is executed. Upon exit from the profiled function, the "_mcount_exit" routine 220 is executed, which causes execution of the "profiler_exit" routine 222 and a return to the caller of the profiled function.

Fig. 4 illustrates an example of a structure of an entry 310 in table 150. For the corresponding profiled function, the entry has a function address field 312, a self time

field 314, a total time field 316, a number of calls field 318, and a Profiler instruction address field 320. Function address field 312 holds the memory address of the first instruction in the profiled function. The self time field 314 holds data indicating the execution time for the profiled function without the execution times for its child functions, if any. The total time field 316 holds data indicating the execution time for the profiled function together with the execution times for its child functions, if any. The number of calls field 318 holds data indicating the number of times the profiled function has been called and executed. The Profiler instruction address field 320 is used for turning profiling on/off for this function.

Fig. 5 illustrates an example of a structure of an entry 410 in stack 130. For the corresponding profiled function, the entry has a return address field 412, a table entry index field 414, an entry time field 416, a re-entry time field 418, and a self time field 420. The return address field 412 holds the instruction address to which control is expected to be returned after the profiled function is executed and is used as described below. The table entry index field 414 holds a pointer to the entry in table 150 for the corresponding profiled function. The entry time field 416 holds data indicating the time that the profiled function was called. The re-entry time field 418 holds data indicating the time that control was returned to the profiled function after execution of a child function of the profiled function. The self time field 420 holds data indicating the execution time for the profiled function without the execution times for its child functions, if any. The contents of fields 416, 418, 420 are used as described below in calculations of the self time and total time described above.

Fig. 7 illustrates an example showing program code that is included in header 212. The top of Fig. 7 lists original program code before Profiler has been added. The bottom of Fig. 7 lists enhanced program code that includes Profiler code 510. An arrow having accompanying comment 'Replaced by "bla _mcount" when Profiler enabled' indicates an instruction location that controls whether the Profiler is enabled. When the location holds instruction "b .+8" as shown in Fig. 7, the Profiler is not enabled, because instruction "b .+8" causes program execution to skip the rest of the header code (the ".long .LPO" instruction), and resume execution with instruction "mr %r31, %r3", which is the first instruction following the Profiler header code.

When "b .+8" is replaced by "bla _mcount", profiling is enabled and the Profiler is called and executed.

In a specific implementation, the "_mcount" routine 214 executes as follows (Fig. 11). All volatile registers are saved (step 1110). A current return address is stored in stack 130 (step 1120). The "profiler_entry" routine 216 is called, with parameters "index" and a pointer to additional parameters (step 1130). Parameter "index" indicates the location, in table 150, of an entry corresponding to the profiled function. The return address of the caller of the "_mcount" routine 214 is replaced with the address of the "_mcount_exit" routine 220 (step 1140). Program execution is returned to the caller of the "_mcount" routine 214 (step 1150).

Thus, according to the specific implementation above, return addresses are manipulated so that the profiled function is called by the "profiler_entry" routine 216 and the "_mcount_exit" routine 220 is executed on the return from execution of body 218.

Fig. 8 illustrates an example showing program code that is included in the "profiler_entry" routine 216, which code executes as follows (Fig. 12). A value representing the current time is recorded (step 1210). A check is performed to determine whether Profiler operation should terminate due to expiration of a pre-set timer (step 1220). A timer is started for the profiled function (i.e., for the "self" time) (step 1230), and a timer keeping track of the total self time is reset (step 1240). The stack index for the profiled function is set (step 1250). The timer for the profiled function's parent function (if any) is halted (step 1260). A number-of calls counter is incremented to update the number of times that the profiled function has been called (step 1270). The self stack frame is pushed onto the profiler stack (step 1280). The profiled function's address and parameters are recorded, if "trace" mode is in effect (step 1290).

In a specific implementation, the "_mcount_exit" routine executes as follows (Fig. 13). The "profiler_exit" routine is called (step 1310). The original return address is retrieved from the Profiler stack (step 1320). The original return address is stored in an "LR" register (step 1330). Control is returned to the original calling function (step 1340).

Fig. 9 illustrates an example showing program code that is included in the "profiler_exit" routine 222, which code executes as follows (Fig. 14). A value representing the current time is recorded (step 1410). The calling function's stack frame is popped from the Profiler stack (step 1420). The re-entry time for the parent function is reset (step 1430). If the Profiler is enabled, the profiled function's Profiler table entry is updated (step 1440).

Fig. 10 illustrates a detailed example of Profiler data structures in use as described above. The return address is address "a0". The address of the profiled function's function

body is address "a1". The address of the return point into the "_mcount" routine from the "profiler_entry" routine is address "a2". The address of the "_mcount_exit" routine is address "a3". The address of the return point into the "_mcount_exit" routine from the "profiler_exit" routine is address "a4". The Profiler stack holds address "a0".

In a specific embodiment, the Profiler is used in three general steps, as follows (Fig. 15). In the first general step, modules are chosen for profiling as now described (step 1510). (In at least some cases, a module with high-level functions is a good candidate.) An emulation is selected. A Profiler compilation option is added to the Makefile entry for the selected module. Listed below is an example showing, in the first line below, a Makefile line lacking the Profiler compilation option, and, in the second line below, a Makefile line having the Profiler compilation option (the points of difference are underlined).

Override modules_r50f+=fastrax_restore.c,:$(code)=cs2;((data)=.data;$(bss)=.bss

Override modules_r50f+=fastrax_restore.c,$(profopts):$(code)=cs2;((data)=.data;$(bss)=.bss

In another embodiment, a "make add_module" procedure may include an explicit profiling option.

In the second general step (step 1520), the Profiler is enabled on one or more address ranges as now described. One or more ranges of addresses are selected to profile (e.g., the entire program code). An "83,def,1,<start>,<end>" utility is invoked (as shown in Fig. 16B as described below). In particular, a "b .+8" instruction is replaced by a "bla _mcount" instruction as described above, in all of the profiled functions' entries. An "83,d,<ranges bitmap>" utility is used (also as shown in Fig. 16B as described below) to disable profiling for ranges of addresses where profiling was previously enabled but currently is not desired.

In the third general step, the Profiler is turned on as now described (step 1530). An ""83,1, <mode>,<time>"" utility is used (as shown in Fig. 16C as described below) to initiate execution of profiling. The "<time>" parameter, which is optional, can be used to cause the profiling to end after a specified amount of time (e.g., a number of seconds) has elapsed. Other utilities "83,A", "83,CA", and "83,0" are used. The "83,A" utility is used to report the status of the Profiler. The "83,CA" utility is used to clear existing statistics. The "83,0" utility is used to stop the execution of profiling.

Fig. 16A illustrates a result of invoking an "83,HELP" utility, which displays a Help main menu. Fig. 16B-16D illustrate a setup help screen, an execution help screen, and a display help screen, respectively. Fig. 17 illustrates a result of invoking the "83,1,PROF" utility which initiates execution of profiling. Fig. 18 illustrates a result of invoking the "83,0" utility which stops execution of profiling. Figs. 19A-19G illustrate a results of invoking an "83,A" utility which displays profiling statistics, including, for each of several profiled functions, the address of the function, the number of times the function has been called, the amount of time elapsed during execution of the function, by itself and including its child functions, and corresponding averages. Fig. 20 illustrates a result of invoking an "83,A,,,SELF,DESC" utility which is a variation of the "83,A" utility, to cause profiling statistics to be displayed in a ranking by field (e.g., "self").

Figs. 21-28 illustrate results of tests performed for an analysis of the performance impact of an example implementation of the Profiler. In particular, Figs. 21-24 illustrate results produced in the case of 4 kilobyte data blocks with program code having embedded but disabled profiler code, and Figs. 25-28 illustrate results produced in the case of 27 kilobyte data blocks with program code having embedded but disabled profiler

code. As shown by the little or no difference between the curves labeled "5567022+ 256 Devs Jml 0" and "5567022+ T11771 v4 256 Devs Jml 0", which represent non profiling and profiling versions of the program code, respectively.

The technique (i.e., one or more of the procedures described above) may be implemented in hardware or software, or a combination of both. In at least some cases, it is advantageous if the technique is implemented in computer programs executing on one or more programmable computers, such as a computer running or able to run Microsoft Windows 95, 98, 2000, Millennium Edition, NT; Unix; Linux; or MacOS; that each include a processor such as an Intel Pentium 4, a storage medium readable by the processor (including volatile and non-volatile memory and/or storage elements), at least one input device such as a keyboard, and at least one output device. Program code is applied to data entered using the input device to perform the method described above and to generate output information. The output information is applied to one or more output devices such as a display screen of the computer.

In at least some cases, it is advantageous if each program is implemented in a high level procedural or object-oriented programming language such as C++, Java, or Perl to communicate with a computer system. However, the programs can be implemented in assembly or machine language, if desired. In any case, the language may be a compiled or interpreted language.

In at least some cases, it is advantageous if each such computer program is stored on a storage medium or device, such as ROM or magnetic diskette, that is readable by a general or special purpose programmable computer for configuring and operating the computer when the storage medium or device is read by the computer to perform the

procedures described in this document. The system may also be considered to be implemented as a computer-readable storage medium, configured with a computer program, where the storage medium so configured causes a computer to operate in a specific and predefined manner.

Other embodiments are within the scope of the invention, and may include one or more of the following features. A warning may be traced if it is detected that excessive time elapses during execution of a function. One or more functions may be traced to build a dependency tree for subsequent use or analysis. Checks may be inserted during profiling, e.g., to log an error if a single function is found to have taken an excessive amount of time (e.g., seconds or minutes) to complete. One or more run time actions can be performed, such as checking stack utilization and issuing an error when the stack becomes full, and checking other dynamic data structures such as the heap.

What is claimed is: